HOW TO CREATE AND MAINTAIN AN AWS SERVERLESS INFRASTRUCTURE WITH TROPOSPHERE AND CODEPIPELINE

AWS CodePipeline | Infrastructure as Code (IaC) | Serverless | Troposphere

beSharp | 6 March 2020

Serverless AWS infrastructures often provide huge advantages with respect to “classic” EC2 based AWS infrastructures as one can easily understand considering, for example, a basic Serverless web application developed using AWS Lambda (backend), DynamoDB (database), Cognito (Authentication) and S3 – CloudFront (Angular single page application). An application like this will scale automatically adapting to every level of traffic, will typically cost significantly less than an EC2 hosted application since the billing will only depend on the traffic and will have a close to zero maintenance cost since you don’t need to update the OS, harden the instance, install security patches and so on. So basically going serverless is a no brainer in most situations: you’ll pay less for an application that will be able to scale faster and you’ll not be responsible anymore for the security of the various application components: AWS will take care of that.

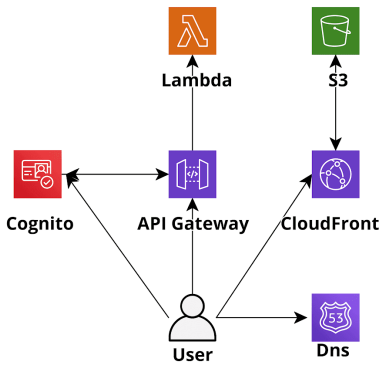


Figure 1: Serverless application example

However, the price you’ll pay to use the serverless paradigm is a significantly more complicated infrastructure: while for a simple EC2 based application you’ll just create an AMI of the instance configuration, create an Autoscaling Group an Elastic Load Balancer and you are good to go, if you

decide to deploy the same application using the serverless paradigm you'll end up with a plethora of Lambda functions, an often tricky cognito configuration, sometimes an API Gateway with a lot of Apis and the DynamoDb configuration. Differently from the code running on an EC2, which is typically self-contained, the serverless code is usually strongly entangled with the AWS infrastructure: if you decide, for example, to add an API endpoint to the backend running on Lambda you'll typically need to create a new function and an API Gateway resource to route requests to the newly created Lambda function.

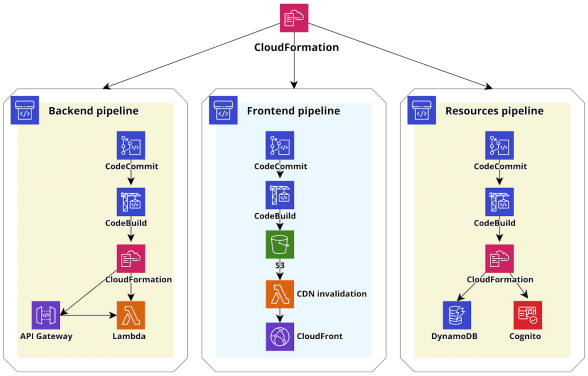
These resources quickly become impossible to manage by hand and thus the creation of an automated CD/CI pipeline is essential to manage the infrastructure. In this setup, once a new commit is pushed in the relevant Git repository branch the AWS CodePipeline triggers, fetches the code from the repository (e.g. AWS Codecommit), builds the code using AWS CodeBuild preparing the Lambda deploy zip packages and an autogenerated cloudformation template. Finally, in the deploy pipeline step, AWS Cloudformation will deploy the template generated in the update step which deploys the new versions of the lambdas and updates the infrastructure accordingly.

While the CloudFormation template taking care of the deployment of the backend could also be used to manage other infrastructure components this should be avoided in most situations. A better way to proceed is instead to create different cloudformation Stacks each taking care of one of the macro building blocks of our infrastructure: for example in the case of a simple serverless infrastructure, like the one we described at the beginning of the post, one could deploy three Cloudformation infrastructure templates: one for the frontend (S3 + CloudFront), one for the backend (API Gateway + Lambda) and one for the other elements in the infrastructure (e.g. DynamoDB and Cognito). Furthermore one could create a further base template to create Account wide resources that will be used by all the templates such as the Cloudtrail configuration and the Vpc Flow Logs configuration. This template should also take care of the creation of the CodePipelines that will deploy the frontend, the backend, and the other needed resources.

Creating a dedicated pipeline to deploy cloudformation templates capable of modifying only the resource needed for the serverless application without the capability to modify anything else is very useful for the developers who thus gain the capability to autonomously change the account configurations in an audited way, without the need to have actual access to the AWS account.

Also, to make life easier for the developers we could use troposphere instead of plain cloudformation (we introduced Troposphere in this article): the syntactic sugar provided by a high-level language is always a great help with respect to plain YAML!

We end up with infrastructure like this one in the figure below:



The YAML template for the Cloudformation step of the backend pipeline can be generated directly by the developers or it can be automatically generated by one of the several frameworks such as Serverless Framework, Chalice, and Zappa. It could also be generated directly from an AWS SAM template: SAM is a simplified templating language specifically designed by AWS to simplify the creation of serverless applications using Cloudformation and can be directly translated into a Cloudformation template using the AWS CLI command AWS cloudformation package.

To conclude we presented a way to build a maintainable AWS infrastructure for serverless applications leveraging the power of AWS CodePipeline, Troposphere and AWS CloudFormation. If you would like to know more about this topic or want to discuss this solution further do not hesitate to reach out to us in the comments section or directly by mail!

beSharp

Dal 2011 beSharp guida le aziende italiane sul Cloud. Dalla piccola impresa alla grande multinazionale, dal manifatturiero al terziario avanzato, aiutiamo le realtà più all'avanguardia a realizzare progetti innovativi in campo IT.

Get in touch

beSharp.it
proud2becloud@besharp.it